

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

Алгоритмы хеширования

По дисциплине: ООП

Обучающийся: Арройо Николас

Факультет: Факультет инфокоммуникационных технологий

Группа: K3220

Направление подготовки: 11.03.02 Инфокоммуникационные технологии и
системы связи

Образовательная программа: Программирование в
инфокоммуникационных системах

Обучающийся

(дата)

(подпись)

Арройо Н.

(Ф.И.О.)

Руководитель

(дата)

(подпись)

Иванов С. Е.

(Ф.И.О.)

ЦЕЛЬ РАБОТЫ

Изучить алгоритмы хеширования, реализовать алгоритм Рабина-Карпа поиска подстроки в строке с применением хеширования.

Упражнение 1. Реализовать поиск одинаковых строк

В этом упражнении было необходимо реализовать программу для поиска одинаковых строк. Основной целью было использование хэширования для группировки одинаковых строк и отображения результатов в упорядоченном виде.

Для упрощения работы, я решил случайным образом сгенерировать 100 строк длиной 4 символа, используя только первые 3 буквы алфавита (a, b, c). Для этого, я создал метод `GenerateRandomString`, который принимает объект класса `Random`, для генерации случайных чисел. В этом методе создается массив символов `stringChars` заданной длины, который заполняется случайными символами, выбранными из константы `chars`, а в конце возвращается сгенерированная строка, как показано на рисунке 1.1.

```
static string GenerateRandomString(int length, Random random)
{
    const string chars = "abc";
    char[] stringChars = new char[length];
    for (int i = 0; i < length; i++)
    {
        stringChars[i] = chars[random.Next(chars.Length)];
    }
    return new string(stringChars);
}
```

Рисунок 1.1 — Метод `GenerateRandomString`

Потом, для генерации массива случайных строк я создал метод `GenerateRandomStrings`, который использует метод `GenerateRandomString`, показан на рисунке 1.1, для создания каждой строки. Этот метод принимает параметры, созданные в методе `Main`, показаны на рисунке 1.3, `n` – количество строк для генерации и `length` – длину каждой строки. Внутри этого метода создается объект `Random`, который предыдущий метод принимает, для

генерации случайных чисел. Еще инициализируется массив строк `strings` с емкостью для `n` элементов, который заполняется случайно сгенерированными строками с использованием метода `GenerateRandomString`. Метод `GenerateRandomStrings` показан на рисунке 1.2.

```
static string[] GenerateRandomStrings(int n, int length)
{
    Random random = new Random();
    string[] strings = new string[n];
    for (int i = 0; i < n; i++)
    {
        strings[i] = GenerateRandomString(length, random);
    }
    return strings;
}
```

Рисунок 1.2 — Метод `GenerateRandomStrings`

В методе `Main` были заданы параметры для генерации строк, которые используются в методе `GenerateRandomStrings`, показан на рисунке 1.2. Все сгенерированные строки были выведены на экран в одной строке с использованием метода `string.Join`, как показано на рисунке 1.3.

```
static void Main(string[] args)
{
    int n = 100;
    int length = 4;

    string[] stringArray = GenerateRandomStrings(n, length);
    Console.WriteLine($"Strings({n}): {string.Join(" ", stringArray)}");
}
```

Рисунок 1.3 — Метод `Main`

Для вычисления хэшей был использован полиномиальный хэш. Для этого были вычислены степени числа $p = 31$ (простое число) до $p^{(length-1)}$, как показано на рисунке 1.4.

```
int p = 31;
int m = length;
int[] p_pow = new int[m];
p_pow[0] = 1;
for (int i = 1; i < m; i++)
{
    p_pow[i] = p_pow[i - 1] * p;
}
```

Рисунок 1.4 — Полиномиальный хэш

Для вычисления хэшей каждой строки был создан словарь `hashs`, где ключом является индекс строки, а значением — вычисленный хэш. В результате для каждой строки получается уникальный хэш, который сохраняется в словаре, как показано на рисунке 1.5.

```
var hashs = new Dictionary<int, int>();
for (int i = 0; i < n; i++)
{
    hashs[i] = 0;
    for (int j = 0; j < stringArray[i].Length; j++)
    {
        hashs[i] += (stringArray[i][j] - 'a' + 1) * p_pow[j];
    }
}
```

Рисунок 1.5 — Словарь `hashs`

После вычисления хэшей для каждой строки, они были выведены на экран в формате номер строки: хэш. Код вывода хэшей показан на рисунке 1.6.

```
Console.WriteLine("\nHash numbers and hashes:");
for (int i = 0; i < n; i++)
{
    Console.WriteLine($"{i + 1} {hashs[i]}");
}
```

Рисунок 1.6 — Код вывода хэшей

Для группировки строк с одинаковыми хэшами был использован метод `GroupBy` из LINQ. Метод `GroupBy` группирует элементы словаря `hashs` по значению хэша (`pair.Value`). Каждая группа содержит строки с одинаковым хэшем. Группы сортируются по первому индексу строки в группе (`group.First().Key`). Результат сохраняется в списке `hashGroups`. Код группировки строк с одинаковым хэшами показан на рисунке 1.7.

```
var hashGroups = hashs.GroupBy(pair => pair.Value)
    .OrderBy(group => group.First().Key)
    .ToList();
```

Рисунок 1.7 — Код группировки строк с одинаковым хэшами

В конце, после группировки строк с одинаковыми хэшами, группы выводятся на экран в формате Group (номер группы): индексы строк, как показано на рисунке 1.8.

```
Console.WriteLine("\nGroups of equal hashes:");
for (int i = 0; i < hashGroups.Count; i++)
{
    Console.WriteLine($"Group {i + 1}: ");
    Console.WriteLine(string.Join(" ", hashGroups[i].Select(pair => pair.Key + 1)));
}
```

Рисунок 1.8 — Код отображения групп одинаковых хэшей

Результаты работы программы показаны на рисунках 1.9, 1.10, и 1.11.

```
Strings(100): cccb acca cabb caba acca aabc bacb cbbb bccc ccbc bacc babb bbba baab cbbe bcac cbca abcc cabb acca
accb acaa cabc bccb baab bccc aaab baac caac bbcc baab baaa bcbc ccac abab abcc cbac cccb acca bbba bbac aabc bacb
cbab abac baba aabc cacc bccb aaab cbcc cacc cbab acba aaac caba cccc acba abcc caaa aaab bcab cacb caab abbc cbb
a babb bbac bccc abac aaba aabb bacb ccaa caab bbbb abaa cbab cbac baab baba accc abbb caac baca cbbe aacb bcac ba
cb cabc bccb bccc acca aacc babb bbba abbb bacb baac bccc
```

Рисунок 1.9 — Случайно сгенерированные строки

```
Hash numbers and hashes:
1 62561
2 32768
3 61538
4 31809
5 32768
6 91327
7 62498
8 61569
9 92351
10 91391
11 92289
12 61537
13 31777
14 60576
15 91360
16 90429
17 32739
18 92319
19 61538
20 32768
21 62559
22 30846
23 91329
24 62560
25 60576
26 92351
27 60575
28 90367
29 90368
30 92320
31 60576
32 30785
33 91390
34 90430
35 60606
36 92319
37 90399
38 62561
39 32768
40 31777
41 90398
42 91327
43 62498
```

Рисунок 1.10 — Часть отображенных хэшей

```

Groups of equal hashes:
Group 1: 1 38
Group 2: 2 5 20 39 93
Group 3: 3 19
Group 4: 4 56
Group 5: 6 42 47
Group 6: 7 43 73 89 98
Group 7: 8
Group 8: 9 26 69 92 100
Group 9: 10
Group 10: 11
Group 11: 12 67 95
Group 12: 13 40 96
Group 13: 14 25 31 80
Group 14: 15 86
Group 15: 16 88
Group 16: 17
Group 17: 18 36 59
Group 18: 21
Group 19: 22
Group 20: 23 90
Group 21: 24 49
Group 22: 27 50 61
Group 23: 28 99
Group 24: 29 84
Group 25: 30
Group 26: 32
Group 27: 33
Group 28: 34
Group 29: 35
Group 30: 37 79
Group 31: 41 68
Group 32: 44 53 78
Group 33: 45 70
Group 34: 46 81
Group 35: 48 52
Group 36: 51
Group 37: 54 58
Group 38: 55
Group 39: 57

```

Рисунок 1.11 — Часть сгруппированных одинаковых хэшей

Упражнение 2. Реализация алгоритма Рабина-Карпа поиска подстроки в строке за $O(N)$.

В этом упражнении было необходимо реализовать программу для поиска всех вхождений подстроки S в тексте T с использованием алгоритма Рабина-Карпа. Основной целью было использование хэширования для эффективного поиска подстроки за время $O(|S| + |T|)$.

Для реализации алгоритма Рабина-Карпа был создан метод `RabinKarp`, который принимает две строки: подстроку S (паттерн) и текст T . В нем предварительно вычисляются степени числа p , создается массив `p_row`, который хранит степени числа p от p^0 до $p^{\max(n,m)-1}$. После этого вычисляются хэша подстроки S с использованием полиномиальной хэш-функции $hash(S) = S[0] \cdot p^0 + S[1] \cdot p^1 + \dots + S[n-1] \cdot p^{n-1}$. Потом, для каждой позиции i в тексте T вычисляется хэш префикса $T[0:i]$, что позволяет быстро

вычислять хэш любой подстроки $T[i:j]$. Эта часть метода показан на рисунке 2.1.

```
static void RabinKarp(string s, string t)
{
    int p = 31;
    int n = s.Length;
    int m = t.Length;

    int[] p_pow = new int[Math.Max(n, m)];
    p_pow[0] = 1;
    for (int i = 1; i < p_pow.Length; i++)
    {
        p_pow[i] = p_pow[i - 1] * p;
    }

    int s_hash = 0;
    for (int i = 0; i < n; i++)
    {
        s_hash += (s[i] - 'a' + 1) * p_pow[i];
    }

    int[] t_hashes = new int[m];
    for (int i = 0; i < m; i++)
    {
        t_hashes[i] = (t[i] - 'a' + 1) * p_pow[i];
        if (i > 0)
        {
            t_hashes[i] += t_hashes[i - 1];
        }
    }
}
```

Рисунок 2.1 — Часть метода RabinKarp

После этого, для каждой возможной подстроки $T[i + n-1]$ вычисляется её хэш и сравнивается с хэшем подстроки S . Если хэши совпадают, выполняется дополнительная проверка на равенство строк (для избежания коллизий). Это часть метода показан на рисунке 2.2.

```
Console.WriteLine("\nPositions where S appears in T:");
for (int i = 0; i <= m - n; i++)
{
    int current_hash = t_hashes[i + n - 1];
    if (i > 0)
    {
        current_hash -= t_hashes[i - 1];
    }

    if (current_hash == s_hash * p_pow[i])
    {
        bool match = true;
        for (int j = 0; j < n; j++)
        {
            if (s[j] != t[i + j])
            {
                match = false;
                break;
            }
        }

        if (match)
        {
            Console.WriteLine(i);
        }
    }
}
```

Рисунок 2.2 — Часть метода RabinKarp

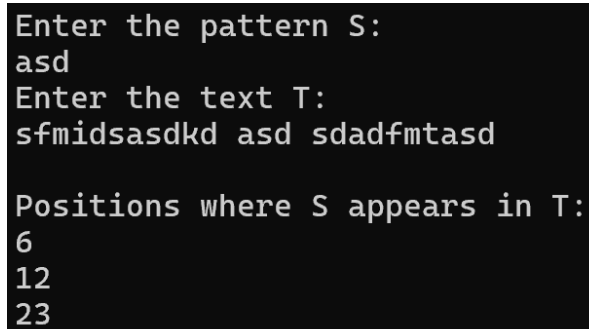
А в методе Main, показан на рисунке 2.3, создаются переменные S и T и вызывается метод RabinKarp, который принимает эти две строки.

```
static void Main(string[] args)
{
    Console.WriteLine("Enter the pattern S:");
    string s = Console.ReadLine();
    Console.WriteLine("Enter the text T:");
    string t = Console.ReadLine();

    RabinKarp(s, t);
}
```

Рисунок 2.3 — Метод Main

Результат запуска программы показан на рисунке 2.4.



```
Enter the pattern S:
asd
Enter the text T:
sfmidsasdkd asd sdadfmtasd

Positions where S appears in T:
6
12
23
```

Рисунок 2.4 — Результат запуск программы

Контрольные вопросы

1. Какие задачи решаются с помощью алгоритма хеширования?

Алгоритмы хеширования применяются для быстрого поиска, хранения и сравнения данных. Они широко используются в структурах данных (например, хеш-таблицы и словари), криптографии, проверке целостности файлов, индексировании и создании уникальных идентификаторов.

2. Как решается проблема коллизий при хешировании?

Коллизии (ситуации, когда разные значения дают одинаковый хеш) решаются с помощью методов разрешения коллизий, таких как цепочки (chaining), открытая адресация (open addressing), двойное хеширование, а также увеличением размера таблицы и использованием более равномерной хеш-функции.

3. Какие ограничения и условия на применение алгоритма хеширования?

Хеширование эффективно только при наличии хорошей хеш-функции, которая равномерно распределяет значения и минимизирует коллизии. Также важно заранее определить допустимую нагрузку на таблицу и предусмотреть обработку коллизий.

4. Какие конструкции ООП С# применяются для реализации алгоритма хеширования?

Используются классы, поля, методы, свойства и коллекции (Dictionary, HashSet). Также может применяться переопределение методов GetHashCode() и Equals() для пользовательских типов, если они используются как ключи в хеш-структурах.

5. Как оценивается сложность алгоритма хеширования?

В среднем, поиск, вставка и удаление выполняются за $O(1)$, но в случае большого количества коллизий или при высокой нагрузке — могут достигать $O(n)$. Эффективность зависит от выбранной хеш-функции и метода разрешения коллизий.

6. Как можно оценить точность алгоритма хеширования?

Точность оценивается по равномерности распределения хешей и количеству коллизий. Чем меньше коллизий и равномернее распределение, тем точнее и эффективнее работает алгоритм. Также можно анализировать нагрузку (load factor) таблицы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы я изучил основы работы с алгоритмами хэширования и их применение для решения задач поиска и группировки данных. В рамках первого упражнения я реализовал программу для поиска одинаковых строк в наборе случайно сгенерированных данных. Использование полиномиальной хэш-функции позволило эффективно группировать строки с одинаковыми значениями, что подтвердило важность хэширования для обработки больших объемов данных. Использование методов GroupBy и OrderBy из LINQ позволило мне упростить процесс группировки и сортировки данных, что является важным навыком для работы с большими наборами информации.

Во втором упражнении я освоил алгоритм Рабина-Карпа, который предназначен для поиска всех вхождений подстроки в тексте. Этот алгоритм продемонстрировал, как хэширование может быть использовано для ускорения поиска подстрок, что особенно полезно при работе с большими текстами. Реализация алгоритма позволила мне понять, как можно эффективно сравнивать строки, используя их хэши, и как избежать коллизий с помощью дополнительной проверки.

Таким образом, лабораторная работа позволила мне не только изучить ключевые алгоритмы хэширования, но и развить навыки программирования на языке C#. Я научился применять хэширование для решения практических задач, таких как поиск одинаковых строк и подстрок, что значительно углубило мое понимание алгоритмов и их применения в реальных проектах.